CS103L SPRING 2020

UNIT 6: POINTERS

TEXT

POINTER MOTIVATION

- Scenario:
  - Collaborating on a writing project. Lots of images. Document ends up being quite large.
- How to share the document?
  - E-mail around a copy of the Word document as an attachment
  - Upload to Google docs and e-mail URL

POINTER MOTIVATION

- These two methods are quite different!
- Google Doc: much less info to send: URL = ~hundred characters, doc = 100MB
- Google Doc: everyone has access to the same document for collaboration
- E-mail: each person has own copy, can edit freely and privately
- We will see this analogy holds well for pointers.
  - URL -> pointer to document

TEXT

POINTER PREVIEW

- Pointers in C++ will let us do the following (details later, of course!)
  - Change the value of a variable declared in one function with a 2nd function
  - Pass large data structures around a program without making copies
    - Making copies is not free!
  - Utilize dynamic memory
    - allocate and de-allocate memory as program runs
  - Interact with hardware based I/O (embedded systems)

ANOTHER POINTER ANALOGY

- Safe Deposit Boxes

0 8	1	2 15	3	4	5 3
6 11	7	8 5	9 7	10 3	11
12	13 1	14	15	16 5	17 3

TEXT

POINTER BASICS

- So what is a pointer?
- Remember:
  - In C++ variables live in memory
  - Each memory location has an address
  - Address is just another way of referring to the location *where* a variable is stored

TEXT

POINTER BASICS

- Pointers are VARIABLES
  - Just like any other variable, has a value and is stored at a location
- Pointers simply store the address of *another* variable
  - Value of a pointer is the memory location of another variable
  - Other variable can be any C++ type: int, double, char… or even another pointer
- Also called references, because they can be used to “refer” to other variables

REFERENCES

- The idea of a reference is very common
  - URL is a reference to a website
  - Phone number is a reference to your phone or VM box
  - Excel sheets: =A1 is a reference in one cell that brings in data from another cell

TEXT

DECLARING POINTERS

- Just like other variable types, we can declare pointer variables
- This gives us a place to *store* the address of something
- Pointers are typed too! Need to know what type the point to.
- int* ptr; (create a pointer to an int)
- char* chr_ptr; (create a pointer to a char)
- double* d; (create a pointer to a double)

POINTER SIZE

- Like all variables, pointers have a size, or number of bits
  - 32-bits on some platforms
  - 64-bits on more modern platforms: Intel, AMD, ARM

TEXT

POINTER OPERATORS

- In C++ we have two operators that let us work with pointers:
- & operator
- * operator

TEXT

& OPERATOR

- & operator is the “address-of” operator
- Returns the address of the variable it touches: &<variable name>
- &x, &y, &start_val, &index, &score
- Returns a pointer!
  - An address is a pointer.
  - Now we can fill our pointer variables:
- int x = 10;
- int* x_ptr = &x

TEXT

* OPERATOR

- * operator: pointer dereferencing operator
- Gives us the data (resolves the reference, aka de-references) at an address (so from a pointer)
- int x = 10;
- int* ptr = &x;
- cout << *ptr << endl;
- What does this print?

& AND * OPERATORS

- & and * are inverses of each other:
- &value => address
- *address => value
- *(&value) => value

TEXT

& AND * HINTS

- Use & when you need to turn a variable into a pointer to that variable
- Use * when you need to get the value that lives at the address contained in a pointer
- Remember: a pointer is a variable, just a special kind that holds the address of another variable

TEXT

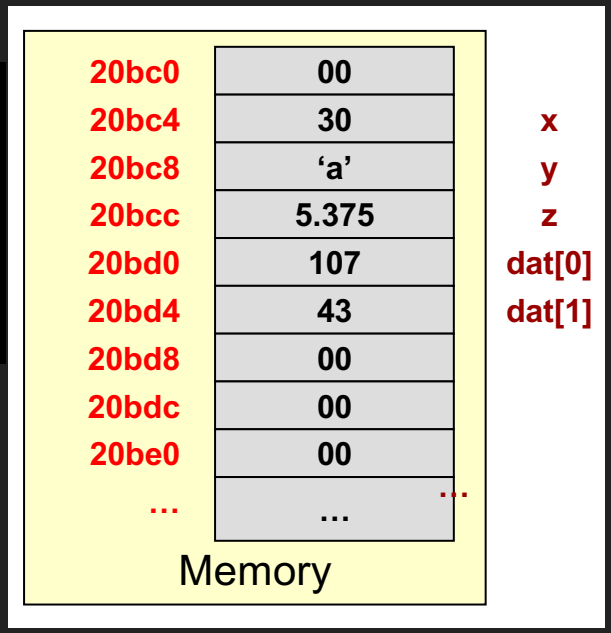
POINTERS AND ARRAYS

- Remember the mantra:
  - int dat[10];
  - What does the variable name ‘dat’ by itself resolve to?
  - The starting address of dat! a.k.a a pointer to dat[0]
  - or sometimes we say “the starting address of the dat array”
  - Both show us ‘dat’ is just a pointer to the array

LOOKING AT SOME POINTERS

- &x = ??
- &y = ??
- &z = ??
- &dat[1] = ??
- dat = ??

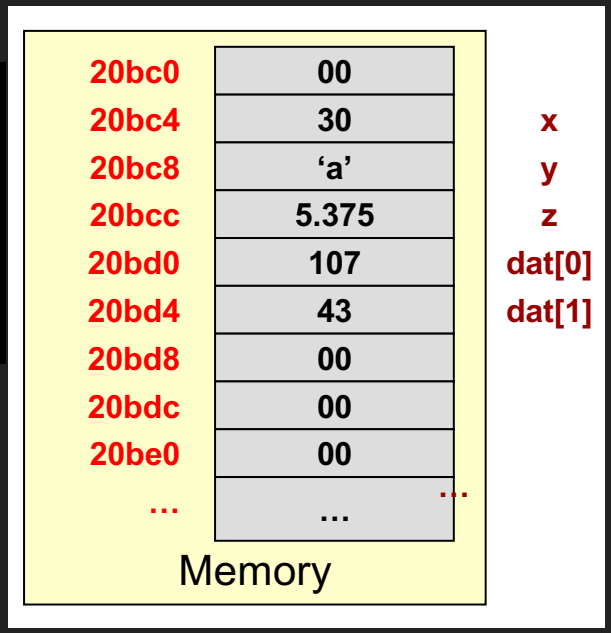
```
int x = 30;
char y='a';
float z= 5.375
int dat[2] = {107, 43};
```



LOOKING AT SOME POINTERS

- &x = 0x20bc4
- &y = 0x20bc8
- &z = 0x20bcc
- &dat[1] = 0x20bd4
- dat = 0x20bd0

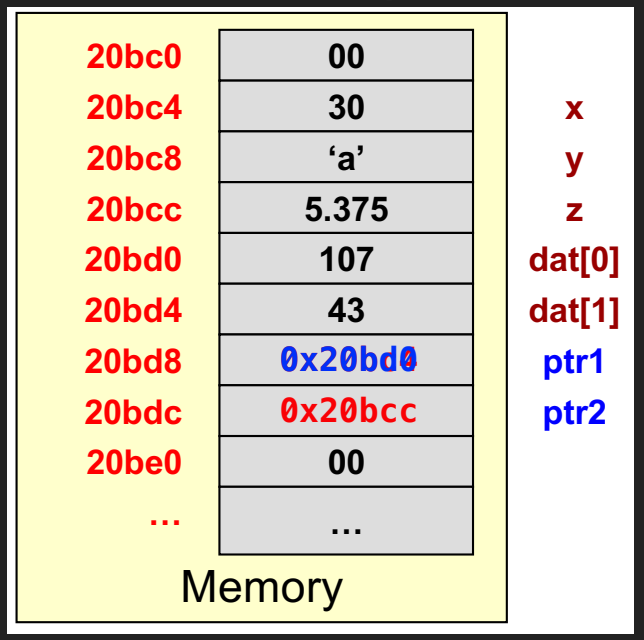
```
int x = 30;
char y='a';
float z= 5.375
int dat[2] = {107, 43};
```



DECLARING POINTERS

- Just like we declare a variable to hold and int, or double we declare a pointer to hold the address of some other variable
- Use the * when declaring to modify the type

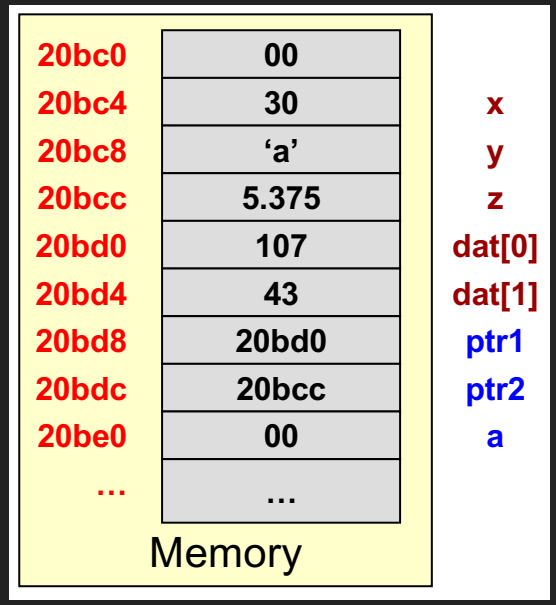
```
int x = 30;
char y='a';
float z= 5.375
int dat[2] = {107, 43};

int *ptr1;
ptr1 = &x; //ptr1 = 0x20bc4
ptr1 = &dat[0]; // ptr1 is now 0x20bd0
//note: this shows you can change the value of a pointer
float* ptr2 = &z;
```



DE-REFERENCING

- Once a pointer points somewhere, we need a way to get the *value* at the destination
- Use the * operator (dereference)
  - *var = “value pointed to by var” or “value at address given by var”

```
ptr1 = dat;
int a = *ptr1 + 5;  a = 112
(*ptr1)++; // *ptr1 = *ptr1 + 1; dat[0] = 108
*ptr2 = *ptr1 - *ptr2;  z = 108 - 5.375 = 102.625
```



TEXT

SYNTAX CHECK

	Declaring a pointer	De-referencing a pointer
char *p		
*p + 1		
int *ptr		
*ptr = 5		
*ptr++		
char *p1[10];		

TEXT

POINTER CHECK POINT

```
int x,y;
int *p = &x;
int *q = &y;
x = 35;
y = 46;
p = q;
*p = 78;
cout << x << " " << y << endl;
cout << *p << " " << *q << endl;
```

TEXT

POINTER/ARRAY REVIEW

- Review:
  - sizeof(int)?
  - sizeof(double)?
  - What does the name of an array evaluate to?
  - If we declare int dat[4] and dat = 0x200, where is dat[1]? dat[2]?

TEXT

POINTER ARITHMETIC

- Pointers are *just* variables that happen to store the address of another variable, an address is just a number
- So we can imagine doing math with the numbers
- int dat[2]; int* ptr = dat;
- ptr++;
- If dat=0x200, which makes more sense:
  - ptr is now 0x201
  - ptr is now 0x204

TEXT

POINTER ARITHMETIC

- Pointer arithmetic shows duality between pointers and arrays
- Adding/subtracting to a pointer adjusts it by the size of the data type
  - int* ptr = dat; ptr++ //increase ptr by 4
  - char* ptr = name; ptr++ //increase ptr by 1
  - double* ptr = data; ptr++ //increase ptr by 8

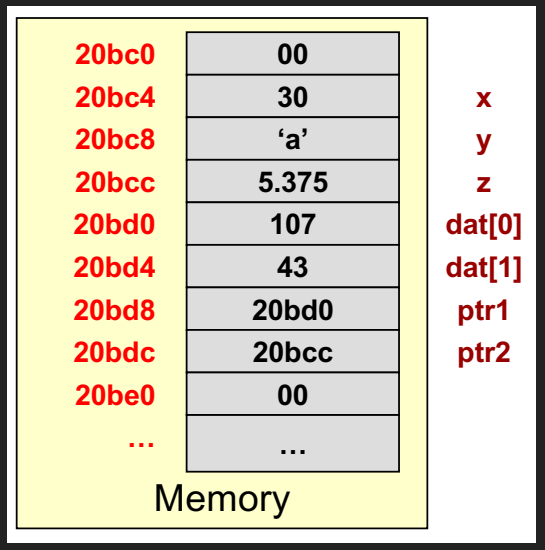
TEXT

POINTER ARITHMETIC

- Why do this strange (but helpful) math?
- What does the name of an array resolve to?
  - The address of the first element = pointer to the first element
  - C/C++ make no distinction between an array and a pointer
- Pointer arithmetic lets you iterate over or skip around an array with math:
  - int dat[10]; int* ptr = dat;
  - cout << dat[i];
  - cout << *(ptr+i);

POINTER ARITHMETIC EXAMPLE

```
ptr1 = dat; //point ptr1 at dat[0]

x = x + *ptr1; //x = 137

ptr1++; //ptr1 now points to dat[1]

x = x + *ptr1; //x = 137 + 43

ptr1 -= 1; //ptr1 now points back to dat[0]
```



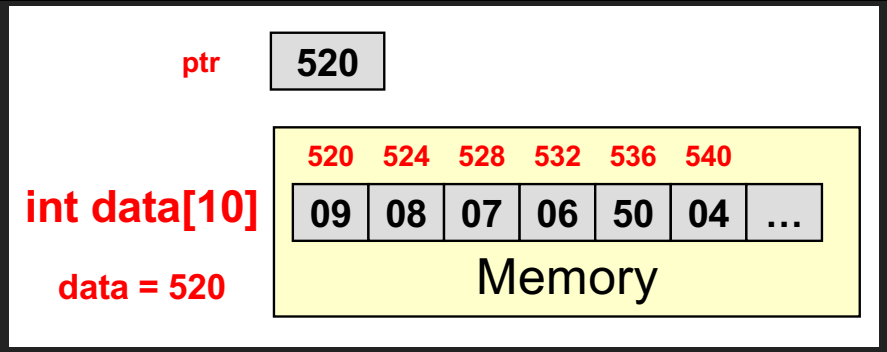
TEXT

POINTERS AND ARRAY INDEXING

- C/C++ make no distinction between pointer arithmetic and array indexing
  - They are equivalent!
- Array syntax: data[i]
  - get the item at the i-th index from the start of the data array)
- Pointer syntax *(data + i)
  - Add i*sizeof() to the data pointer and dereference to get value
- There are exactly the same operation!

MORE POINTER/ARRAY EXAMPLES

```
int data[10]; //data lands at memory address 520
*(data+4) = 50; //data[4] = 50;
int* ptr = data; //now ptr can be used to access data array
ptr[1] = ptr[2] + ptr[3]; //same as data[1] = data[2]+data[3]
```



ANOTHER POINTER/ARRAY EXAMPLE

```
int main(int argc, char *argv[])
{
  int data[10] = {9,8,7,6,5,4,3,2,1,0};

  int* ptr, *another;  // * needed for each
                       //  ptr var. you
                       //  declare

  ptr = data;      // ptr = start address
                   //       of data

  another = data; // another = start addr.

  for(int i=0; i < 10; i++){
    data[i] = 99;
    ptr[i] = 99;  // same as line above
    *another = 99; // same as line above
    another++;

  }

  int x = data[5];
  x = *(ptr+5);  // same as line above
  return 0;

}
```

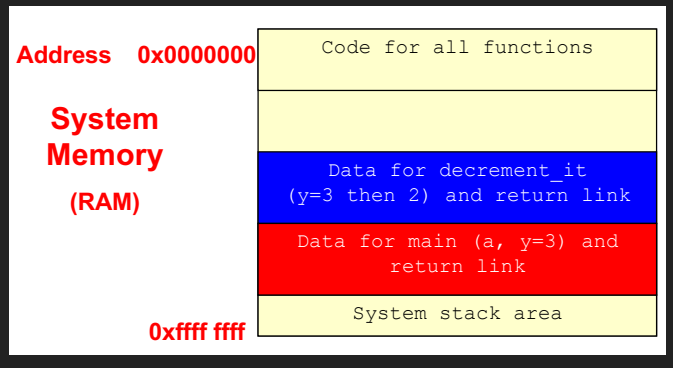
TEXT

SO WHY USE POINTERS?

- Reason #1 (from slide 4)
  - “Change the value of a variable declared in one function with a 2nd function”
  - Known as pass-by-reference
  - We got a preview with passing arrays (which are just pointers!)

PASS BY REFERENCE

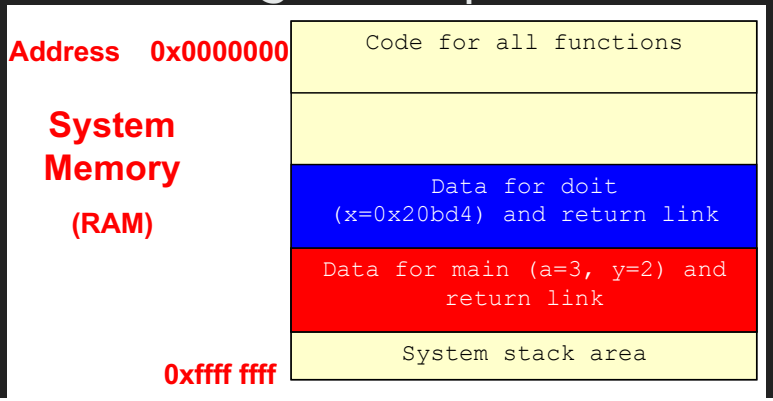
- We know this doesn't work
- Why?
  - Pass-by-value
- A *copy* of the actual parameter is passed into the formal parameter




```
void decrement_it(int);

int main()
{
  int a, y = 3;
  decrement_it(y);
  cout << "y = " << y << endl;
}

void decrement_it(int y)
{
   y--;
}
```

PASS-BY-POINTER (AKA PASS-BY-REFERENCE)

- Rewrite to use pointers. Now it works.
- Pointer value (memory address) is still passed-by-value
- But we can dereference the memory address to manipulate the value!
- The *value* of y in main() is changed and persists after decrement_it() runs




```
int main()
{
  int a, y = 3;
  // assume y @ 0x20bd4
  // assume ptr
  a = y;
  decrement_it(&y);
  cout << "a=" << a;
  cout << "y=" << y << endl;
  return 0;
}


// Remember * in a type/
// declaration means "pointer"
// variable
void decrement_it(int* x)
{
   *x = *x - 1;
}
```

HERE COMES SWAP AGAIN!

- Task: write function to swap the value of two variables.
- We know pass-by-value won't work.
  - Swap would be performed on variables *local* to the function
- With pointers, we can implement swap
  - Memory address are passed in and dereferenced to perform swap

```
int main()
{
  int x=5,y=7;
  swapit(x,y);
  cout << "x=" << x << " y=";
  cout << y << endl;
}

void swapit(int x, int y)
{  int temp;
   temp = x;
   x = y;
   y = temp;
}
```

program output: x=5,y=7

```
int main()
{ int x=5,y=7;
  swapit(&x,&y);
  cout << "x=" << x << "y=";
  cout << y << endl;
}

void swapit(int *x, int *y)
{  int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

program output: x=7,y=5

PASSING ARRAYS AS ARGUMENTS

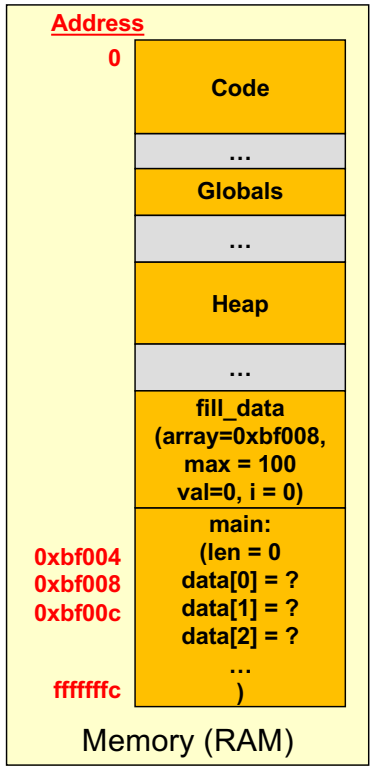
- We've seen this in "arrays" lecture, now we understand even better
- Remember C/C++ doesn't keep track of the size for us, so *most* functions that take arrays will also take a size parameter

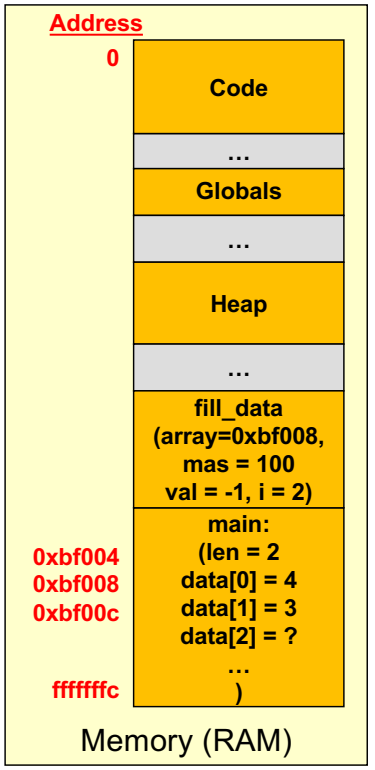
```
void add_1_to_array_of_10(int []);
void add_1_to_array(int *, int);
int main(int argc, char *argv[])
{
  int data[10] = {9,8,7,6,5,4,3,2,1,0};
  add_1_to_array_of_10(data);
  cout << "data[0]" << data[0] << endl;
  add_1_to_array(data,10);
  cout << "data[9]" << data[9] << endl;
  return 0;
}

void add_1_to_array_of_10(int my_array[])
{
  int i=0;
  for(i=0; i < 10; i++){
    my_array[i]++;
  }
}
void add_1_to_array(int *my_array, int size)
{
  int i=0;
  for(i=0; i < size; i++){
    my_array[i]++;
  }
}
```

ARGUMENT PASSING EXAMPLE

- Now we can write helpful functions that do useful things on arrays

```
#include <iostream>
using namespace std;

int main()
{
  int len=0;
  int data[100];

  len = fill_data(data, 100);

  for(int i=0; i < len; i++)
    cout << data[i] << " ";
  cout << endl;
  return 0;
}

// fills in integer array w/ int's
// from user until -1 is entered
int fill_data(int *array, int max)
{
   int val = 0;
   int i = 0;
   while(i < max){
     cin >> val;
     if (val != -1)
       array[i++] = val;
     else
       break;
   }
   return i;
}
```





TEXT

IN CLASS EXCERCISES

- Swap
- Roll2
- Product

TEXT

POINTERS TO POINTERS

- Say what?
- As if pointers are not confusing enough!
- Extend our safe deposit box analogy

POINTERS TO POINTERS ANALOGY

- A box might contain gold
- Or it contains a box-id (and presumably the key)
- You can follow the chain of box-ids until you find the gold
- No limit to the # of indirections
  - *9 = gold in box 7 (9 -> 7)
  - **16 = gold in box 3 (16 -> 5 -> 3)
  - ***0 = gold in box 3 (0 -> 8 -> 5 -> 3)

0 8	1	2 15	3	4	5 3
6 11	7	8 5	9 7	10 3	11
12	13 1	14	15	16 5	17 3

TEXT

POINTERS TO POINTERS

- In C/C++ we can do the same thing.
- Remember, pointers are variables.
- They just happen to hold the memory address of another variable.
- Now they just happen to hold the memory address of another variable that also happens to hold the memory address of...(like inception)

TEXT

LEVELS OF INDIRECTION

- Since we can have as many levels of indirection as we want, we need to tell compiler how many there are when declaring the pointers:

```
int *p; //pointer to an int, one hop away
double **q; //pointer to a double, two hops away
```

POINTERS TO POINTERS

- So now we can create pointers to pointers, what does that look like?

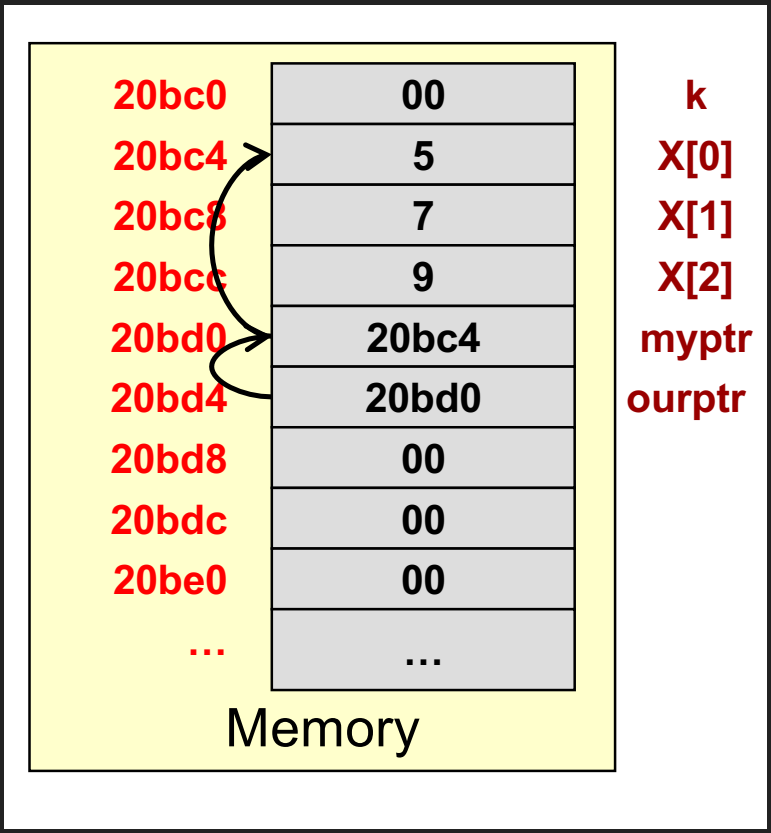
```
int k, x[3] = {5,7,9};
int *ptr1, **ptr2;
ptr1 = x;
ptr2 = &ptr1;
k = *ptr1;
k = (**ptr2) + 1;
k = *(*ptr2 + 1);
```

Declaration	Expression	Yields
int *ptr1	*ptr	int
int **ptr2	**ptr2	int
	*ptr2	int*

POINTERS TO POINTERS

- Follow the chain…

```
int k, x[3] = {5,7,9};
int *ptr1, **ptr2;
ptr1 = x;
ptr2 = &ptr1;
k = *ptr1; //k = 5
k = (**ptr2) + 1; //k = 6
k = *(*ptr2 + 1); //k = 7
```



POINTERS TO POINTER REVIEW

- Consider the declarations:

```
int k, x[3] = {5,7,9};
int *ptr1, **ptr2;
```

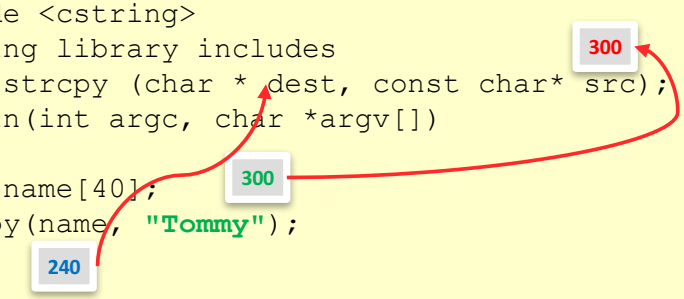
Expression	Type
x[0]	
x	
ptr1	
*ptr1	
*ptr2	
ptr1+1	
ptr2	

C-STRING CONSTANTS

- Review: text in “double quotes” is a string constant
- The end up being an array of NULL terminated char’s (c-string)
- Assigned memory address
- If you check, the type is const char*
  - You cannot/should not *change* these strings (hence the const)

```
int main(int argc, char *argv[])
{
  // These are examples of C-String constants
  cout << "Hello" << endl;
  cout << "Bye!" << endl;
  ...
}
```

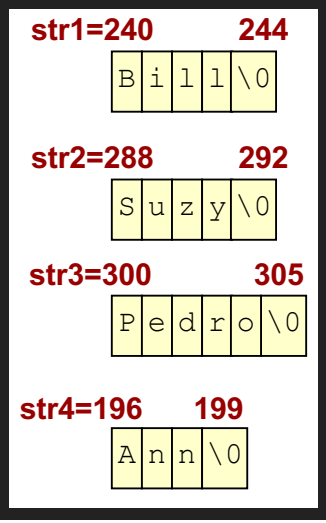

```
#include <cstring>
//cstring library includes
//void strcpy (char * dest, const char* src);
int main(int argc, char *argv[])
{
  char name[40];
  strcpy(name, "Tommy");
}
```



ARRAYS OF POINTERS AND C-STRINGS

- Sometimes need to store several arrays
  - The arrays are related, like names
  - C-strings are arrays of char variables
  - Is this a good way to do it?

```
int main(int argc, char *argv[])
{
  int i;
  char str1[] = "Bill";
  char str2[] = "Suzy";
  char str3[] = "Pedro";
  char str4[] = "Ann";

  // I would like to print out each name
  cout << str1 << endl;
  cout << str2 << endl;
  ...
}
```

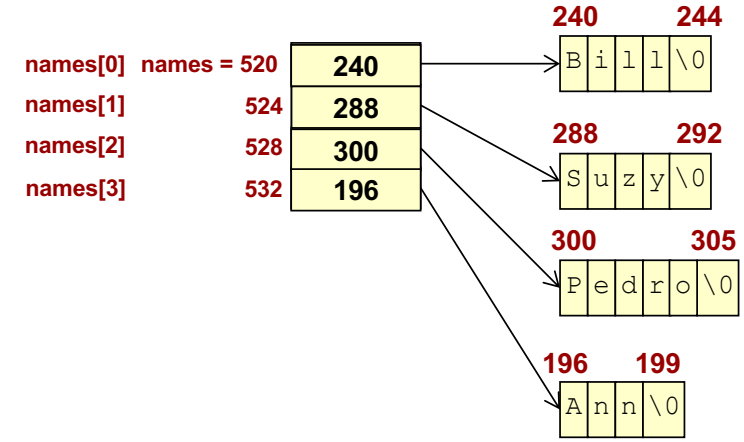


ARRAYS OF POINTERS TO C-STRINGS

- Is this better?
  - Somewhat?
- What does names resolve to?
  - The address of the first char*
  - names has what type? char**
- What type does names[0] have?
  - char*

```
int main(int argc, char *argv[])
{
  int i;
  char str1[] = "Bill";
  char str2[] = "Suzy";
  char str3[] = "Pedro";
  char str4[] = "Ann";
  char *names[4];

  names[0] = str1; ...; names[3] = str4;

  for(i=0; i < 4; i++){
    cout << names[i] << endl;
  }
  ...
}
```

Still painful

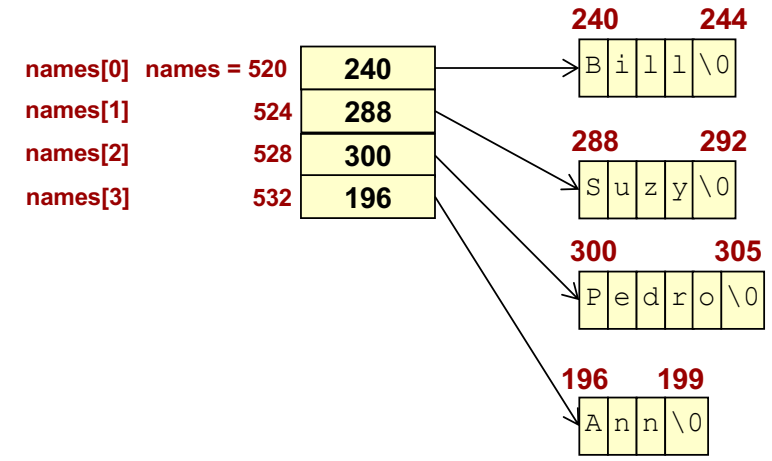


ARRAYS OF POINTERS

- Remember a pointer is just another variable, so we can have arrays of pointers (just like we could have an array of ints)

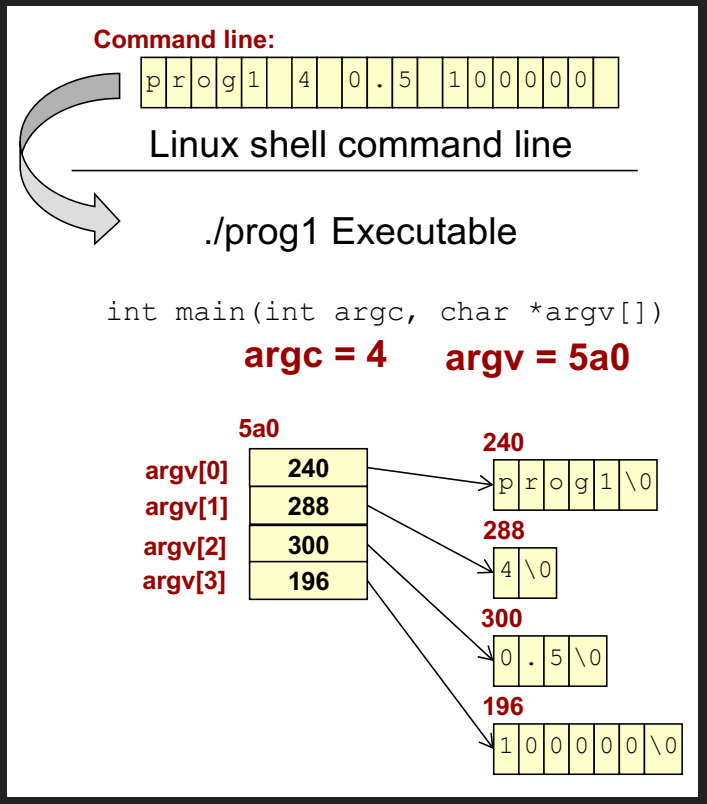
```
char *names[4] ={"Bill",
                 "Suzy",
                 "Pedro",
                 "Ann"};

int main(int argc, char *argv[])
{
  int i;

  for(i=0; i < 4; i++){
    cout << names[i] << endl;
  }
  return 0;
}
```

Painless?!?



NOW WE CAN UNDERSTAND COMMAND LINE ARGUMENTS

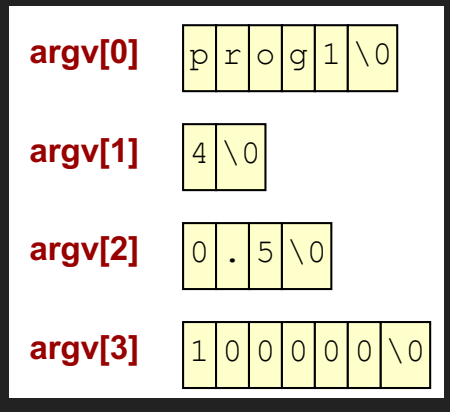
- When we run a command, the shell gives our program some information:
  - int argc -> #of command line arguments
  - char* argv[] -> array argc# of pointers to C-strings
- The command line is broken down by whitespace, each piece gets a C-string, and a spot in argv[]
- argv[0] is always program name
- Example:
  - ./prog1 4 0.5 10000
  - ./zombies 20 2 10000 137957



TYPICAL COMMAND LINE USAGE

- Twentyone assignment uses command line args
- At beginning of main(), check to see if argc is correct
  - Implies the user put in the right # of arguments
- Remember, each argument is a C-string
  - String arguments = OK
  - Numeric arguments: need to convert
    - int atoi() (ASCII to Int)
    - double atof() (ASCII to Floating point)

```
#include <iostream>
#include <cstdlib>
using namespace std;

// char **argv is the same as char *argv[]
int main(int argc, char **argv)
{
  int init, num_sims;
  double p;
  if(argc < 4){
    cout << "usage: prog1 init p sims" << endl;
    return 1;
  }

  init = atoi(argv[1]);
  p = atof(argv[2]);
  num_sims = atoi(argv[3]);

  ...
```



STRING FUNCTION LIBRARY

- #include <cstring> gets you lots of useful C-string functions
- int strlen(char *dest);
- int strcmp(char *s1, char*s2);
  - Return 0 if equal, >0 if s1 would come lexicographically before s2, <0 otherwise
- char* strcpy(char *dest, char *src);
  - Copy src to dest
- char* strcat(char *dest, char *src);
  - Copy src to the end of dst and NULL terminate. Aka concatenate src onto dest
- char* strchr(char *str, char c);
  - Return a pointer to the first occurrence of c in str, NULL otherwise

IN CLASS EXERCISES

- cmdargs_sum
- cmdargs_smartsum
- cmdargs_smartsum_str
- toi

TEXT

WHY USE POINTERS?

- To change a variable(s) local to one function in another function ✔
- a.k.a. pass-by-pointer or pass-by-reference
- When using large data structures, pass by pointer avoids making copies ✔
  - Copies aren't free: memory use and time
- When we need a variable address. We don't know, or can't know the address of a desired memory address at compile time ←
- Accessing embedded hardware registers

TEXT

DYNAMIC MEMORY – MOTIVATION

- Writing a program to calculate student grades.
- int scores[??];
  - How many students?
- The following is bad form/unsupported
  - int num; cin >> num; int scores[num];
- Also, variables declared on the stack die when that function finishes…
- How to solve this problem?

DYNAMIC MEMORY – ANALOGY

- Public storage
- Need more space? Go rent a locker.
  - Need more? Rent another.
- Had a garage sale, don’t need them any more?
  - Terminate the lease.
  - Locker becomes available to someone else

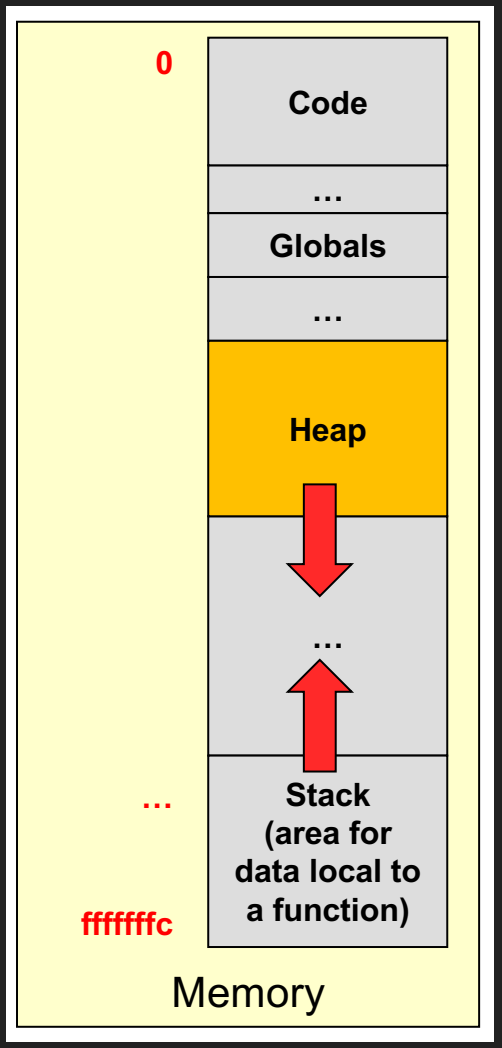
TEXT

DYNAMIC MEMORY

- Dynamic memory is on-demand-memory
- When you need some memory you allocate what you need
  - This memory is allocated on the heap
- When you are done, you release it back to the heap
- Heap memory is visible (valid) any where in your program
  - So any function with a pointer to heap-allocated memory can use it

THE HEAP

- The heap is an area of memory that can grow/shrink as your program executes
  - i.e. it is “dynamic”
- Stack grows towards lower addresses (up on the drawing)
- Heap grows towards higher addresses (down)
- In rare cases they collide - PANIC ENSUES
  - or your program crashes



TEXT

C++ DYNAMIC MEMORY ALLOCATION

- ‘new’ operator
  - Use to allocate memory from the heap. Either single variables or arrays

```
double *dptr = new double; //allocate 8 bytes to hold one double
int* array = new int[100]; //allocate an array of 100 integers
```

  - ‘new’ returns a pointer to the newly allocated memory
  - The pointer is of the appropriate type

TEXT

C++ DYNAMIC MEMORY DEALLOCATION

- `delete` operator gives the memory back
  - Delete is a misnomer! Nothing is deleted or zero'd out
- Calling `delete` on a pointer makes the memory available again, so it can be reused

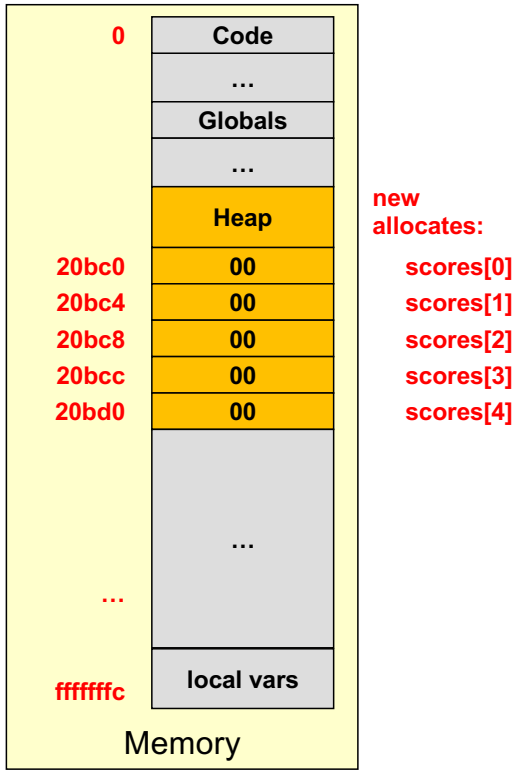
```
delete dptr; //give back the 8 bytes

delete[] array; //give back the memory allocated for the entire array
```

- Note the syntax [] for arrays

DYNAMIC MEMORY USAGE

```
int main(int argc, char *argv[])
{
  int num;

  cout << "How many students?" << endl;
  cin >> num;

  int *scores = new int[num];
  // can now access scores[0] .. scores[num-1];
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int num;

  cout << "How many students?" << endl;
  cin >> num;

  int *scores = new int[num];
  // can now access scores[0] .. scores[num-1];
  delete [] scores
  return 0;
}
```



TEXT

DYNAMIC MEMORY CHECKPOINT

```
_________ data = new int;

_________ data = new char;

_________ data = new char[100];

_________ data = new char*[20];

_________ data = new string;
```

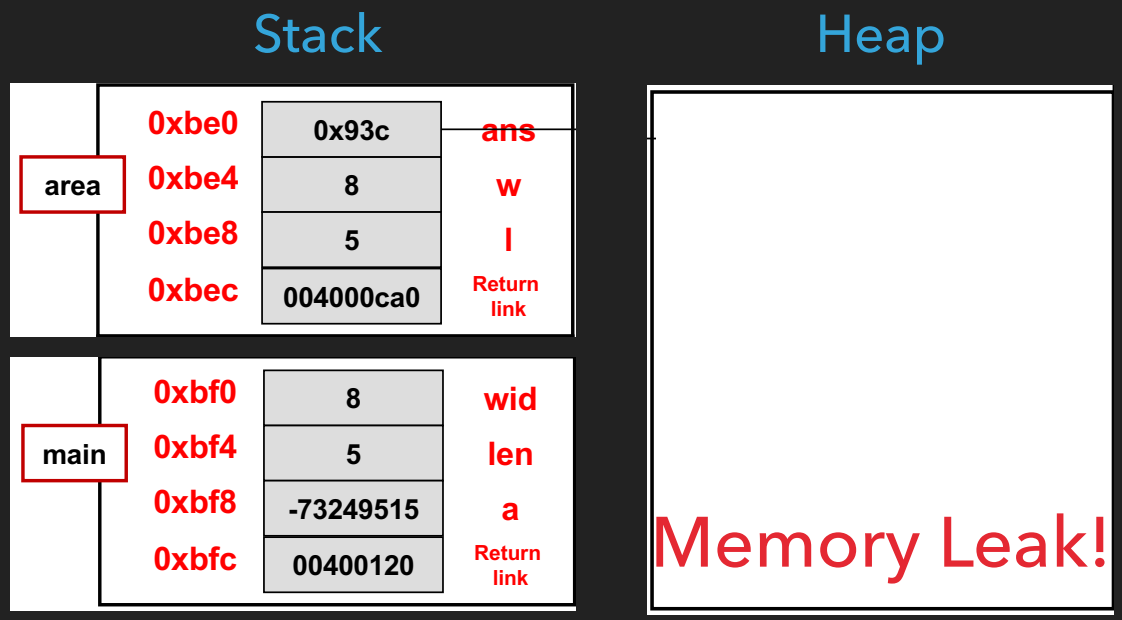
TEXT

DYNAMIC MEMORY CHECKPOINT

```
int* data = new int;

char* data = new char;

char* data = new char[100];

char** data = new char*[20];

string* data = new string;
```

DYNAMIC MEMORY USAGE

- Dynamic Memory 'lives' on heap
- Only accessible with the pointer returned from new




```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len);
}


int* area(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```

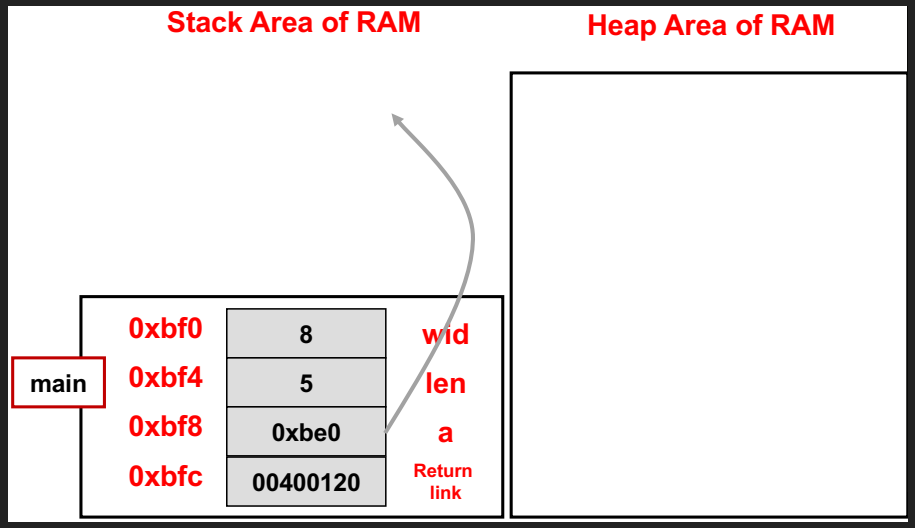
MEMORY LEAKS

- You must keep track of heap allocated pointers!
- If you don't, the memory "leaks"
  - Allocated, but no way to deallocate
- Now this code can delete the memory when finished with it

```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);
int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
  delete a;
}

int* area(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```

POINTER TIP!

- Never return a pointer to a local variable!




```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
}


int* area(int w, int l)
{
  int ans;
  ans = w * l;
  return &ans;
}
```

IN CLASS EXERCISE

- ordered_array

TEXT

MULTIDIMENSIONAL ARRAYS AND DYNAMIC MEMORY

- How to allocate 2D, 3D or more arrays?
- int* image = new int[x][y]; //This won't compile
- Solution? Higher dimensions are allocated as pointers, only lowest dimension is data type.
- This is how the wordBank[] in lab 7 works.

ALLOCATING MULTIDIMENSIONAL ARRAYS

- Example: allocate image array X, Y

```
int** image = new int*[X];

for(int i=0;i<X;i++)

{

    image[i] = new int[Y];

}

//Then the syntax to use is the same

image[i][j] = 0;
```

ALLOCATING 3D EXAMPLE

```
int*** array = new int**[X];
for(int i=0;i<X;i++)
{
    array[i] = new int*[Y];
    for(int j=0;j<Y;j++)
    {
        array[i][j] = new int[Z];
    }
}
//now we can use it
array[1][2][3] = 0;
```